GD2S02: Software Engineering for Games

Design Patterns

What are design patterns?

- *"Patterns identify and specify abstractions that are above the level of single classes and instances, or of components."*

  Definition given by the Gang of Four.

- The most compelling motivation behind identifying patterns is to separate things that change from things that stay the same.
  - Following this principle reduces the effort to make modifications on parts that could actually be unchanged – which causes lower costs and makes programs easier to understand.

Design Patterns

- Each pattern has four essential elements:
  1. Pattern name:

     A word or two describing the design problem, its solution, and consequences.
  2. Problem:

     The design problem and its context, when to apply the pattern.
  3. Solution:

     The elements that make up the pattern, their relationships, responsibilities and collaborations.
  4. Consequences:

     Results of using the pattern.

Design Patterns

- Why do we use Design Patterns?
  - Makes it easier to reuse successful designs and architectures.
  - Makes proven techniques more accessible to developers of new systems.
  - Helps the developer choose design alternatives that make a system reusable and avoid alternatives that compromise reusability.
  - Can improve the documentation and maintenance of existing systems by furnishing an explicit specification of class and object interactions and their underlying intent.

Design Patterns

- Why do we use Design Patterns?
  - Design patterns can speed up the development process by providing tested, proven development paradigms.
  - Reusing design patterns helps to prevent subtle issues that can cause major problems, and it also improves code readability for coders and architects who are familiar with the patterns.
  - Put simply, design patterns help a designer get a design "right" faster.

The Gang of Four – Principles of OO Design

"Program to an 'interface', not an 'implementation'."

- Do not declare variables to be an instance of a particular concrete class. Commit only to an interface defined by an abstract class.
- Utilise creational patterns instead (Factory, Singleton, etc).

The Gang of Four – Principles of OO Design

"*Favor object composition over class inheritance*."

- Because inheritance exposes a subclass to details of its parent's implementation, it is often said that 'inheritance breaks encapsulation'.
- Object composition is defined dynamically at run-time through objects acquiring references to other objects.

Inheritance

```
class Object
{
  public:
     virtual void update() {};
     virtual void draw() {};
     virtual void collide(Object objects[]) {};
};

class Visible : public Object
{
  public:
     virtual void draw() { /* draw model at position of this object */ };
  private:
     Model* model;
};

class Solid : public Object
{
    public:
        virtual void collide(Object objects[]) { /* check and react to collisions with objects */
};
};

class Movable : public Object
{
    public:
        virtual void update() { /* update position */ };
};
```

Inheritance

- Then, we have concrete classes:
  - Class Player - which is Solid, Movable and Visible
  - Class Cloud - which is Movable and Visible, but not Solid
  - Class Building - which is Solid and Visible, but not Movable
  - Class Trap - which is Solid and neither Visible nor Movable
- Either do multiple inheritance (“diamond problem”)
- or make classes
  - like VisibleAndSolid,
  - VisibleAndMovable,
  - VisibleAndSolidAndMovable, etc.
  - For every needed combination, which leads to a large amount of repetitive code.

Composition

```
class Object
{
    public:
        Object(VisibilityDelegate *v, UpdateDelegate *u, CollisionDelegate *c) : _v(v), _u(u), _c(c) {};

        void update() { _u->update(); };
        void draw()   { _v->draw(); };
        void collide(Object objects[]) { _c->collide(objects); };
    private:
        VisibilityDelegate *_v;
        UpdateDelegate *_u;
        CollisionDelegate *_c;
};

class VisibilityDelegate
{
    public:
        virtual void draw() = 0;
};
 class Invisible : public VisibilityDelegate
{
    public:
        virtual void draw() {};
};
 class Visible: public VisibilityDelegate
{
    public:
        virtual void draw() { /* draw model */ };
};
```

Composition

```
class CollisionDelegate
{
    public:
        virtual void collide(Object objects[]) = 0;
};
class Solid : public CollisionDelegate
{
    public:
        virtual void collide(Object objects[]) { /* check collisions with object and react */ };

class NotSolid : public CollisionDelegate
{
    public:
        virtual void collide(Object objects[]) {};
};

class UpdateDelegate
{
    public:
        virtual void update() = 0;
};

class Movable : public UpdateDelegate
{
    public:
        virtual void update() { /* move object */ };
};

class NotMovable : public UpdateDelegate
{
    public:
        virtual void update() { };
};
```

Then, concrete classes would look like:

```
class Player : public Object
{
    public:
        Player():Object(new Visible(), new Movable(), new
Solid()) {};
        […]
};

class Smoke : public Object
{
    public:
        Smoke():Object(new Visible(), new Movable(), new
NotSolid()) {};
        […]
};
```

Design Pattern Space

		Purpose		
		Creational	Structural	Behavioural
Scope	class	Factory Method	Adapter (class)	Interpreter Template Method
	object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Façade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Design Patterns are classified by two criteria:
1- Purpose
2- Scope

Design Pattern Space

- Purpose reflects what the pattern does, it can be:
  1. Creational
     - Concerned with the process of object/class creation.
  2. Structural
     - Concerned with the composition of classes/objects.
  3. Behavioural
     - Concerned with the different ways classes and objects interact and distribute responsibilities.
- Scope specifies whether the pattern is applied to objects or classes.
  - Class patterns are static-fixed at compile time, while object patterns can be changed at run-and are more dynamic.

Creational Design Patterns

- Creational patterns abstract the instantiation process.
- The creational patterns aim to separate a system from how its objects are created, composed, and represented.
- Object creational patterns depend more on object composition than class inheritance.
- They increase the system's flexibility in terms of the what, who, how, and when of object creation.

Examples of Creational Design patterns

- Factory Method (Class creational)
  - Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
- Abstract Factory
  - Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
- Builder
  - Separate the construction of a complex object from its representation so that the same construction process can create different representations.

Examples of Creational Design patterns

- Prototype
  - Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
- Singleton
  - Ensure a class only has one instance, and provide a global point of access to it.

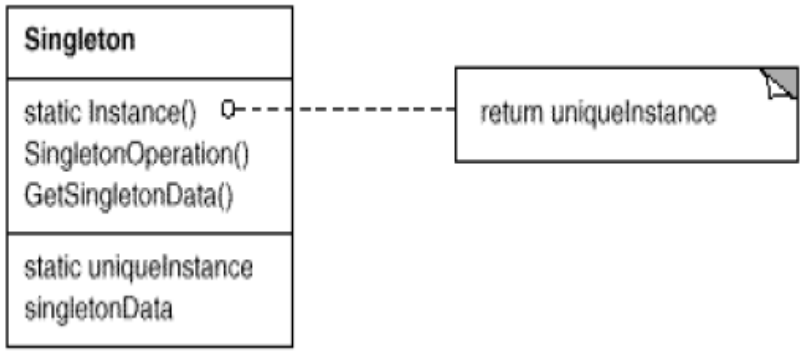
Singleton

- Object-Creational design pattern.
- The aim that a class has only one instance, with a global point of access to it.
- Provides one single object
  - It instantiates automatically for the first access and gives for the second and all other following accesses only the reference from the first instantiated object.
  - Example
    - Spooler inside a printing system.
    - Window-manger from MS-Windows.

Singleton

- Use the Singleton pattern when:
  - There must be exactly one instance of a class, and it must be accessible to clients from a well-known access point.
  - When the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code.
- Clients access a Singleton instance solely through Singleton's Instance operation.

- Structure



Singleton

- Define all constructors to be protected or private, to ensure that only one instance can ever get created.
- Clients access the singleton **exclusively** through the *Instance* member function.
- The variable _instance is initialized to 0.
- The static member function *Instance* returns its value, initializing it with the unique instance if it is 0.
- Instance uses lazy initialization; the value it returns isn't created and stored until it's first accessed.

```
class Singleton {
public:
        static Singleton* Instance();
protected:
        Singleton();
private:
        static Singleton* _instance;
};

Singleton* Singleton::_instance = 0;
Singleton* Singleton::Instance () {
   if (_instance == 0) {
        _instance = new Singleton;
    }
    return _instance;
}
```

Structural Design Patterns

- Concerned with how classes and objects are composed to form larger structures.
- Structural class patterns use inheritance to compose interfaces or implementations. This is particularly useful for making independently developed class libraries work together.
- Structural *object* patterns describe ways to compose objects to realize new functionality.

Examples of Structural Design patterns

- Adapter (class, object)
  - Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
- Bridge
  - Decouple an abstraction from its implementation so that the two can vary independently.
- Composite
  - Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

Examples of Structural Design patterns

- Decorator
  - Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
- Façade
  - Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
- Flyweight
  - Use sharing to support large numbers of fine-grained objects efficiently.
- Proxy
  - Provide a surrogate or placeholder for another object to control access to it.

Flyweight Design Pattern

- Uses sharing to support large numbers of fine-grained objects efficiently without prohibitive cost.
- The **flyweight design pattern** allows you to reuse memory spaces in an application when you have lots of objects that are almost identical in nature.
  - For example, if you are writing a game for a Smartphone where the amount of memory is very limited and you need to show many aliens that are identical in shape, you can have only one place that holds the shape of the alien instead of keeping identical shape in the precious memory.

Flyweight Design Pattern

- A **Flyweight** is a shared object that can be used in multiple contexts simultaneously.
- In the **flyweight pattern**, there is the concept of Intrinsic and Extrinsic state.
- Intrinsic states
  - Are stored in the flyweight; it consists of information that's independent of the flyweight's context, thereby making it sharable.
- Extrinsic states
  - Depend on and varies with the flyweight's context and therefore can't be shared.

Flyweight Design Pattern

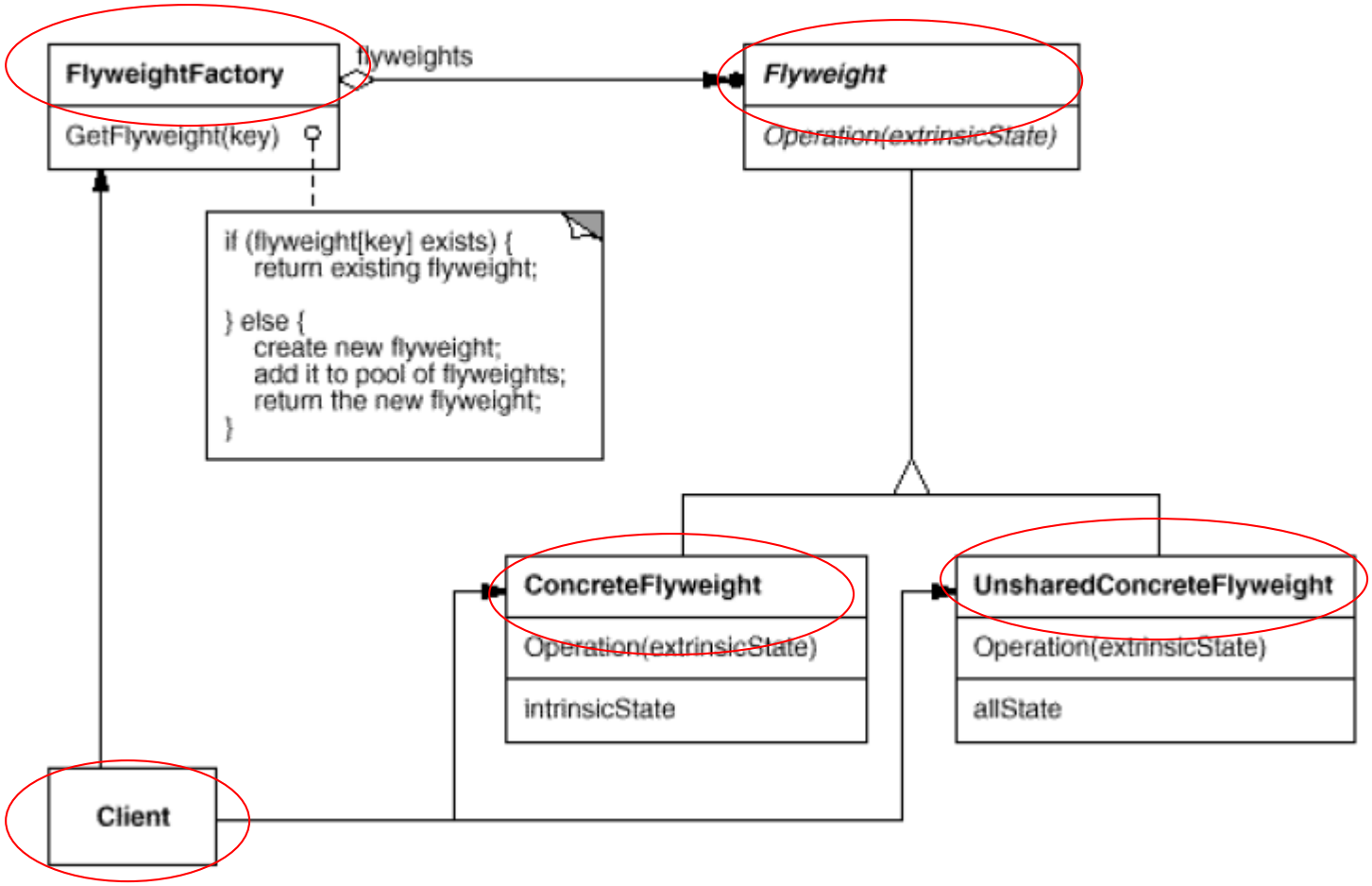
- For example, in the game that we would like to create, the shapes of the aliens are all the same, but their color will change based on how mad each one is.
- Intrinsic State
  - The shapes of the aliens
- Extrinsic State
  - The color of the aliens.

Flyweight Design Pattern

- When to use Flyweight Design Pattern?
  - SW uses a large number of objects.
  - Storage costs are high.
  - Most object state can be made extrinsic.
  - Many groups of objects may be replaced by relatively few shared objects once extrinsic state is removed.
  - The application doesn't depend on object identity.

Flyweight Design Pattern

- **UML** of the **flyweight pattern**



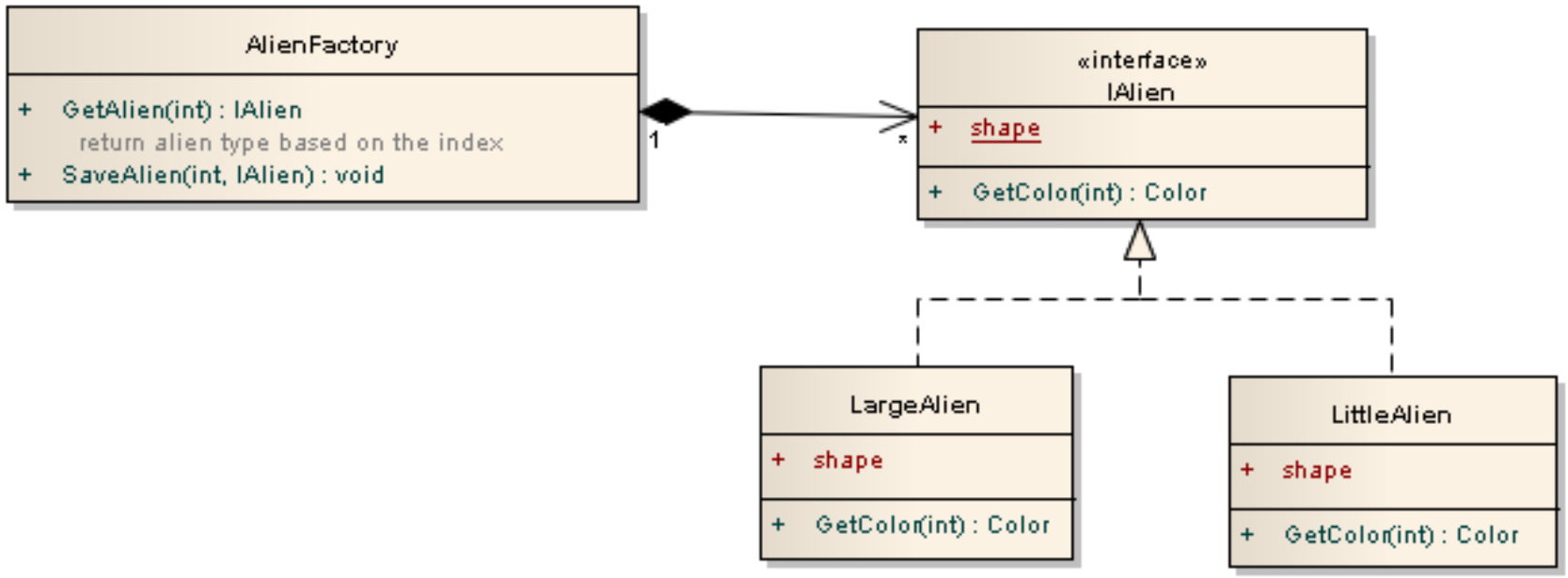
Flyweight Design Pattern

- **Flyweight**
  - An interface through which flyweights can receive and act on extrinsic state.
- **FlyweightFactory**
  - Creates and manages flyweight objects.
  - Ensures that flyweights are shared properly.
- **Client**
  - Maintains a reference to flyweight(s).
  - Computes or stores the extrinsic state of flyweight(s).
- **ConcreteFlyweight**
  - Implements the Flyweight interface and adds storage for intrinsic states.
  - A ConcreteFlyweight object must be sharable.
- **UnsharedConcreteFlyweight**
  - Not all Flyweight subclasses need to be shared. It's commo[illegible] UnsharedConcreteFlyweight objects to have ConcreteFlyweight obje[illegible]s as children at some level in the flyweight object structure.

How to Develop Flyweight

- Remove extrinsic states.
  - Remove the non-shareable state from the class attributes, and add it to the calling argument list of affected methods.
- The client must use the Factory instead of the new operator to request objects.
- The client (or a third party) must look-up or compute the non-shareable state, and supply that state to class methods.

Apply it to our example...

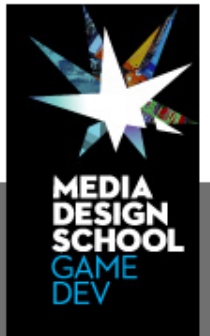


- The AlienFactory class stores and retrieves different types of aliens using its GetAlien and SaveAlien methods.
- The IAlien interface defines the shape property as the intrinsic state and the GetColor method as the extrinsic state.
- The LargeAlien and the LittleAlien classes are the **flyweight** objects where each own shape intrinsic state and ways of calculating the GetColor extrinsic state.

Code Example

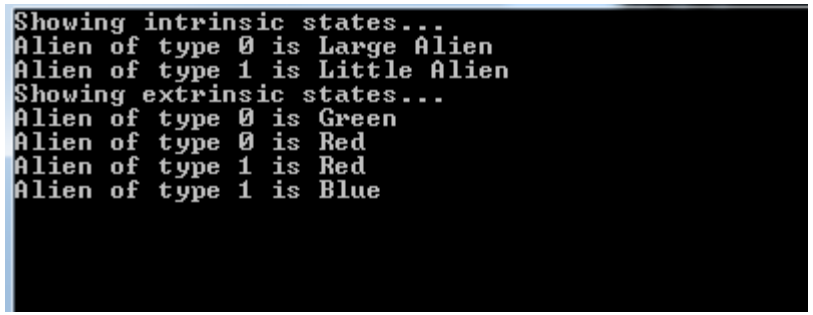
```
enum Color { Green, Red, Blue };
char *Colors[] = {"Green","Red","Blue"};
class IAlien
{
public:
                virtual Color GetColor(int) = 0;   //extrinsic state
                void SetShape(char* shape)
                  {  Shape = shape;   }
                char* GetShape()
                  {    return Shape;  }
protected :
                char* Shape;  //intrinsic state
};

class AlienFactory
{
private:
                IAlien* list [10];
public:
                void SaveAlien(int index, IAlien* alien)
                {
                     list[index] = alien;
                }
                IAlien* GetAlien(int index)
                {
                    return list[index];
                }
};
```

```
class LargeAlien : public IAlien
{
public:
                Color GetColor(int madLevel)    //extrinsic state
                {
                  if (madLevel == 0)
                                return Green;
                  else if (madLevel == 1)
                                return Red;
                   else
                                return Blue;
                }

};

class LittleAlien : public IAlien
{
public:
                Color GetColor(int madLevel)    //extrinsic state
                {
                  if (madLevel == 0)
                                return Red;
                  else if (madLevel == 1)
                                return Blue;
                  else
                                return Green;
                }
};
```

Code Example

```
void main()
{
//create Aliens and store in factory
AlienFactory factory ;
LargeAlien*  Lrgalien= new LargeAlien;
Lrgalien->SetShape("Large Alien");
LittleAlien* Ltlalien = new LittleAlien;
Ltlalien->SetShape("Little Alien");
factory.SaveAlien(0, Lrgalien);
factory.SaveAlien(1, Ltlalien);
//now access the flyweight objects
IAlien* a = factory.GetAlien(0);
IAlien* b = factory.GetAlien(1);
//show intrinsic states, all accessed in memory without calculations
cout<<"Showing intrinsic states..." << endl;
cout << "Alien of type 0 is " << a->GetShape()<< endl;
cout << "Alien of type 1 is " << b->GetShape() << endl;

//show extrinsic states, need calculations
cout << "Showing extrinsic states..." << endl;
cout << "Alien of type 0 is " << Colors[a->GetColor(0)] << endl;
cout << "Alien of type 0 is " << Colors[a->GetColor(1)] << endl;
cout << "Alien of type 1 is " << Colors[b->GetColor(0)] << endl;
cout << "Alien of type 1 is " << Colors[b->GetColor(1)] << endl;
}
```



Behavioral Design Patterns

- Behavioral patterns describe patterns of objects or classes and the patterns of communication between them.
- They are concerned with algorithms and the assignment of responsibilities between objects.
- Behavioral class patterns use inheritance to distribute behavior between classes.
- Behavioral object patterns use object composition rather than inheritance.

Behavioral Design Patterns Examples

- Interpreter (Class)
  - Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.
- Template Method (Class)
  - Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure

Behavioral Design Patterns Examples

- Chain of Responsibility
  - Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.
- Command
  - Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
- Iterator
  - Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Behavioral Design Patterns Examples

- Mediator
  - Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.
- Memento
  - Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.
- Observer
  - Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

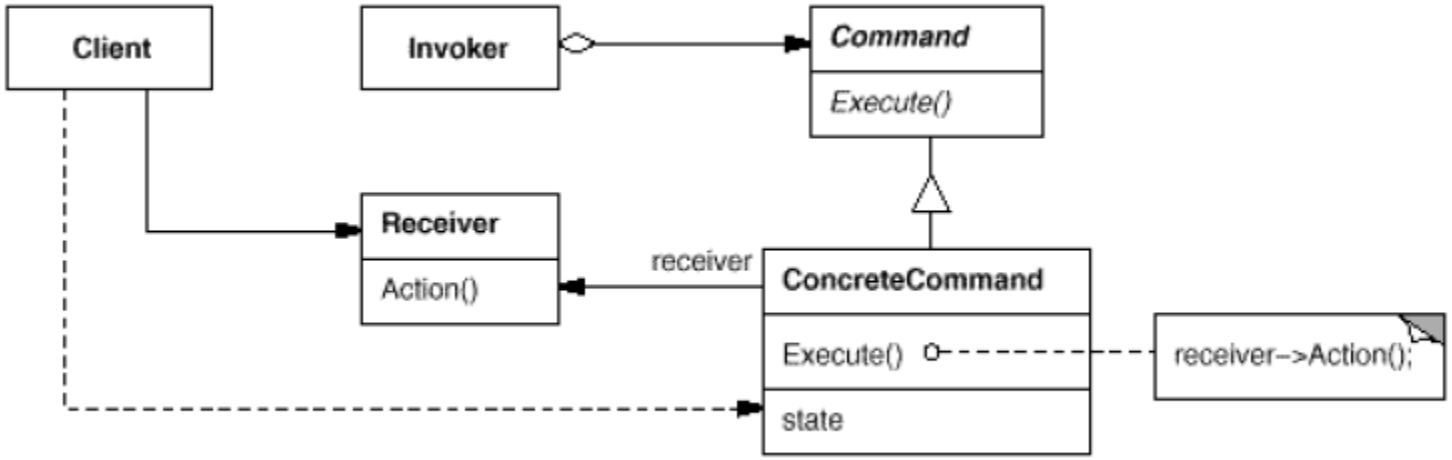
Behavioral Design Patterns Examples

- State
  - Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
- Strategy
  - Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
- Visitor
  - Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates

Command Design Pattern

- Object behavioral pattern.
- Encapsulates requests as objects.
- Allows to parametrize clients with different requests, queue, or log requests.
- Decouples the object that invokes the operation from the one that performs it.

UML Diagram of a Command Pattern

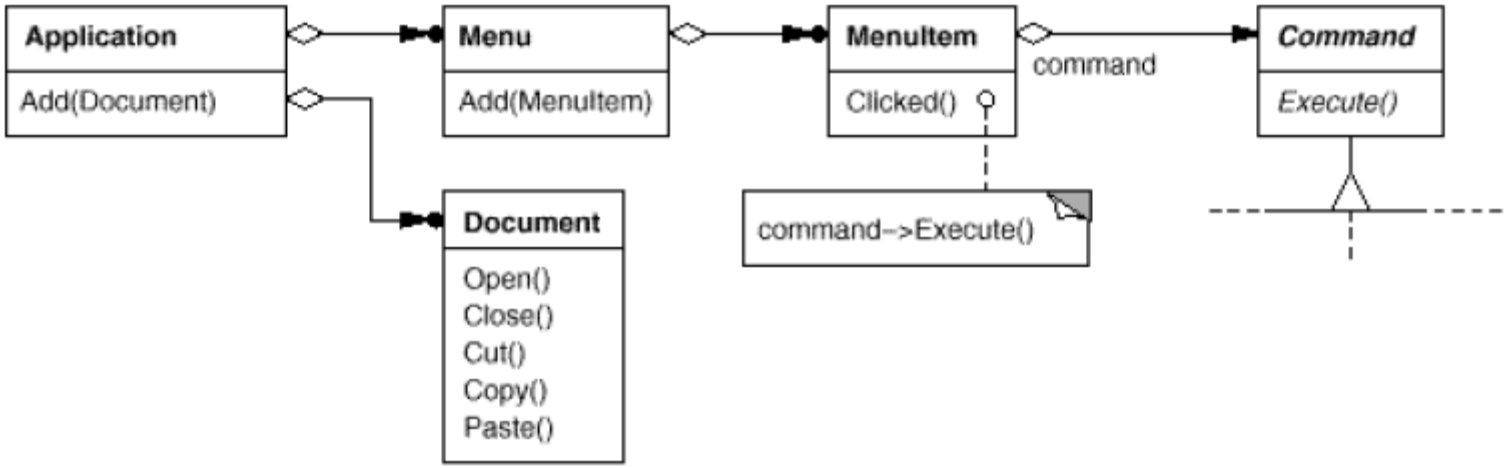


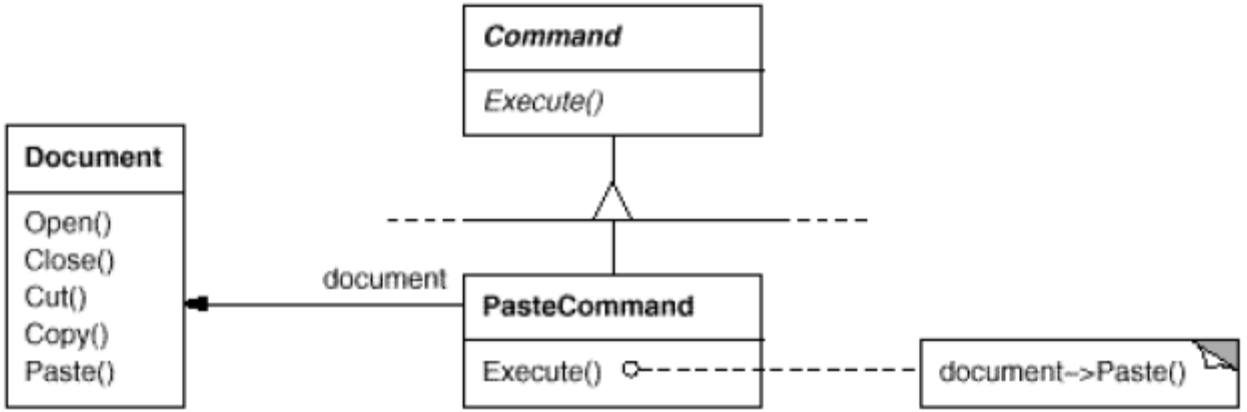
- Command
  - Interface for executing an operation.
- ConcreteCommand
  - Defines a binding between a Receiver object and an action.
  - Implements Execute by invoking the corresponding operation(s) on Receiver.

UML Diagram of a Command Pattern

- **Client**
  - Creates a ConcreteCommand object and sets its receiver.
- **Invoker**
  - Asks the command to carry out the request.
  - Stores the ConcreteCommand Object.
- **Receiver**
  - Knows how to perform the operations associated with carrying out a request. Any class may serve as a Receiver.

Command Design Pattern Example





MEDIA DESIGN SCHOOL GAME DEV

Command Design Pattern Example

```
class Command {
public:
	virtual ~Command();
	virtual void Execute() = 0;
protected:
	Command();
}
```

A PasteCommand must be passed a Document object as its receiver. The receiver is given as a parameter to PasteCommand's constructor.

```
class PasteCommand : public Command {
public:
	PasteCommand(Document*);
	virtual void Execute();
private:
	Document* _document;
}
PasteCommand::PasteCommand (Document* doc){ _document = doc;
Void PasteCommand::Execute () { _document->Paste(); }
```

Summary

- Design patterns
- Types
- Creational
  - Singleton
- Structural
  - Flyweight
- Behavioural
  - Command